Cloud Application Performance Monitoring

Chandra Krintz, Rich Wolski, Hiranya Jayathilaka, and Wei-Tsung Lin

Computer Science Department
University of California, Santa Barbara

1. ABSTRACT

In this paper, we overview the design and implementation of a new approach to Application Performance Monitoring (APM) for Cloud Platforms-as-a-service (PaaS). Our approach couples and integrates full stack performance monitoring and analysis into the PaaS system itself for comprehensive introspection. To enable this, we employ lightweight and intelligent sensors and agents and "pluggable" data analysis modules that facilitate service level objectives (SLOs) for application response time, application-specific performance anomaly detection and root cause analysis, and workload change point detection. We implement our APM by combining the popular Elastic Stack with other common PaaS services in a way that is portable so that it can be integrated easily into public and private PaaS systems.

2. INTRODUCTION

Over the past decade Platform-as-a-Service (PaaS) has become a popular approach used by enterprises, institutions, and developers for deploying web-accessible applications in the cloud [25]. The PaaS abstraction effectively hides execution details such as physical resource allocation (CPU, memory, disk etc), service ecosystem management, the operating system, and the network configuration. PaaS clouds also automate load balancing and resource scaling, and provide high availability and fault tolerance of PaaS components and services. As a result, PaaS enables application developers to focus on the programming and innovation aspects of their applications, without having to be concerned about deployment and system issues.

The wide spread use of PaaS technology has intensified the need for new techniques to monitor applications deployed in a PaaS cloud. Monitoring is key for facilitating effective auto-scaling and resource utilization, for maintaining high availability, for managing multi-tenancy (multiple applications sharing resources), and for identifying performance bugs in both the applications and the PasS itself. To extract sufficient operational insight in order to drive this functionality requires that the performance data collected be extensive and comprehensive. At the same time, collection of this data must be low-overhead and not perturb the performance and behavior of the system significantly. Thus the key to any successful Application Performance Monitoring (APM) system must effectively navigate this tension and its associated trade-offs.

Toward this end, we propose a new design for PaaS APMs that can be easily integrated within most PaaS systems. Our Cloud APM is not an external system that monitors a PaaS cloud from the outside (which most APM systems do today [22, 5, 7]). Rather, it integrates with the PaaS cloud thereby leveraging and augmenting the existing components of the PaaS cloud to provide comprehensive, full-stack monitoring, analytics, and visualization capabilities. We believe that this design choice is a key differentiator over existing PaaS and cloud application monitoring systems because it enables us to take advantage of the scaling, efficiency, services, fault tolerance, security, and control features that PaaS systems currently offer, while providing scalable, low overhead end-to-end monitoring and analysis of cloud applications.

This paper overviews our Cloud APM architecture and describes its PaaS integration. We discuss the individual components of our Cloud APM and how they interact. Where appropriate, we also detail the concrete technologies (tools and products) that we use to implement various components of the Cloud APM, and provide our rationale and intuition behind the use of these technologies.

2.1 Cloud APM Design and PaaS Integration

Like most system monitoring solutions, our Cloud APM implements four primary functions: data collection, storage, processing (which includes data analytics), and visualization. Data collection is performed by various sensors and agents that instrument the applications and the core components of the PaaS cloud. Sensors in our system are primitive in their capability to monitor a given component. Agents in contrast are more complex and as a result are able to intelligently adapt to changing conditions, making dynamic decisions about what information to capture and when and how to capture it. The implementation of sensors and agents impact PaaS behavior and performance and thus must be as lightweight and non-intrusive as possible while capturing behavior and performance accurately. To achieve this, we combine sampling (periodic, non-exhaustive measurement) and intelligent placement of sensors and agents throughout the software stack.

For storage and processing of performance data, we leverage the scalable, durable, and highly available distributed services of the PaaS itself. In particular, our APM system makes use of scalable key-value stores, caching systems, relational database systems, high performance search systems, as well as batch and streaming analytics frameworks that make up the service ecosystem of the PaaS. To keep our APM portable across PaaS systems, each function defines an application programming interface (API) through which it interacts with the components and services of the PaaS. To port our APM to a new PaaS, we rewrite the API operations to link to those of the PaaS.

Figure 1 illustrates our APM integration with a typical PaaS stack. The left (dark blue) boxes depict the PaaS architecture. Arrows indicate the flow of data and control in response to application requests. At the lowest level of a PaaS cloud is an infrastructure layer that consists of the necessary compute, storage and networking resources that the PaaS acquires and releases dynamically. The PaaS kernel is a collection of managed, scalable services that implement common functionality that most cloud applications (apps) require for their functionality. Application developers implement their innovations by composing these services. The services of extant PaaS systems typically include data storage and caching, queuing services, authentication and user management services and many others.

Increasingly, PaaS clouds provide a managed set of APIs (typically called a cloud software development kit (SDK)) that are used by developers to link functionality that the PaaS provides into their applications. CloudSDKs (like other similar PaaS proxy mechanisms) simplify, control, and load balance access to PaaS services across applications and the system [9, 20, 16].

Application servers execute (copies of) an application and link the application code with and the underlying PaaS kernel. The servers also isolate and sandbox application code for secure, multi-tenant operation and to facilitate control over and charging for service use. Load balancers (or frontends) serve as the entry point to applications. They intercept application requests, filter, and route them to appropriate application server instances for handling.

Our Cloud APM components integrate into a PaaS at different points in the PaaS architecture. We depict APM components in grey with their interactions denoted by the black lines in the figure. The small grey boxes attached to the PaaS components represent the sensors and agents we employ to monitor components of the cloud platform (including the application) to collect events and performance data. Note that the APM collects data from all layers in the PaaS stack (i.e. it performs full stack monitoring). The rate at which measurements are taken is configurable and in many cases adaptive (e.g. depending on the percent overhead introduced by each agent). Moreover, our sensors and agents batch operations and perform asynchronous communication to reduce performance perturbation and overhead.

From the frontend and load balancing layer, the APM gathers information related to incoming application requests. Our monitoring agents scrape HTTP server logs to extract timestamps, source/destination addresses, response time, and other HTTP message parameters. This information is readily available for harvesting in most technologies employed today as frontend technologies (e.g. Apache HTTPD, Nginx). Additionally, our agents collect information pertaining to active connections, invalid access attempts, and HTTP errors.

Within the application server layer, APM sensors collect application and runtime/container log data. Such data typically includes process level metrics indicating the resource usage of the individual application instances. By targeting log files, we avoid the high overhead of application and application server instrumentation. This can be added via additional agents if the benefits they bring (e.g. more accurate information) warrant their overhead.

Within the PaaS kernel, we instrument the entry points of *all* PaaS services. We collect the caller and callee (target operation) information, a timestamp, measured execution time per operation invocation, and details about the request including size and a hash of the arguments. This information helps us distinguish different phases of PaaS execution and to aggregate and characterize operation invocation instances. We use these details to avoid instrumenting the application for profiling purposes (enabling low overhead yet accurate full stack monitoring).

From the cloud infrastructure, we collect information related to virtual machines, operating system containers and processes, and the resource usage of individual physical host machines. Most of our APM sensors at this level scrape logs and query the Linux `proc` file system. We gather metrics about network usage, CPU and memory use, as well as decisions about resource allocation and reclamation made by resource management and orchestration frameworks (e.g. Kubernetes [**?**], Mesos [**?**, **?**], Yarn [**?**]).

Each of our sensors and agents collect information that enables us to cluster related system activities and events across the system. Given that PaaS systems commonly host web applications and services, our APM design considers web requests as events. Our system tags all incoming requests with unique identifiers (a feature available to most modern frontend technologies). We configure the system to attach a request identifier to each HTTP request header, which is visible to all components. We then configure the appropriate APM agents to record these identifiers when recording an event. Our data processing layer then clusters measurements by request identifiers to facilitate end-to-end system analysis for individual web requests.

Our data processing layer stores and provides scalable access to this performance data. It also permits "plugging in" of analysis routines that can be used to characterize application and system behavior over time, to detect behavioral and performance anomalies and workload changes, and to identify opportunities for more effective resource utilization and automatic scaling of resources, services, and application instances. These analysis routines perform both inference and prediction and make use of statistical analysis libraries, batch processing services (MapReduce [**?**], Spark [**?**], machine and deep learning systems [**?**, **?**, **?**], etc.), and search/query systems [8, **?**].

3. IMPLEMENTATION

We next outline some of the technologies and tools that we have chosen to implement our APM architecture. After a thorough evaluation of numerous existing system monitoring tools and platforms, we choose Elastic Stack [8] as our APM foundation. Elastic Stack is an open source distributed system consisting of ElasticSearch, Logstash, and Kibana, among other tools. We use ElasticSearch for APM

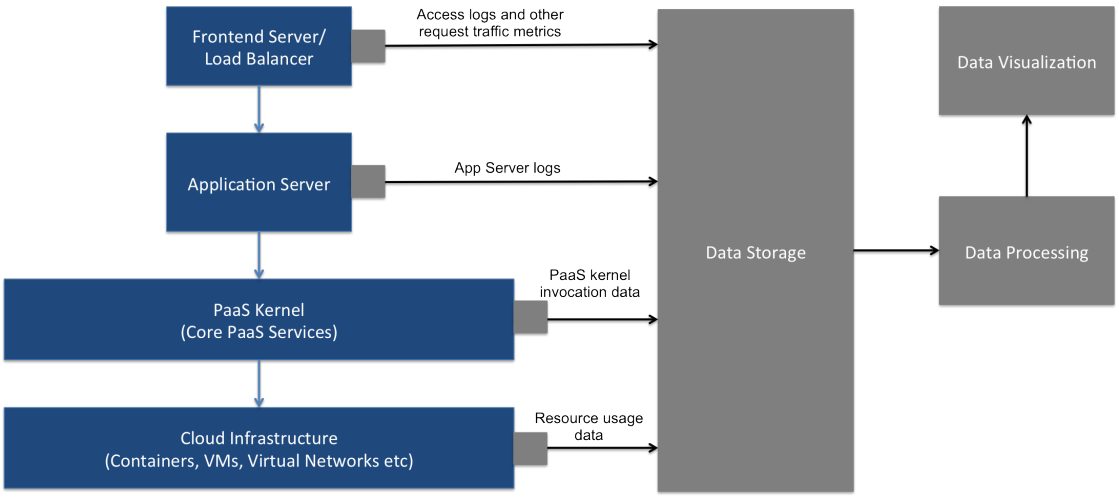

Figure 1: APM architecture.

data storage and processing. ElasticSearch supports scalable and highly available management of structured and semi-structured data like that collected by our agents and sensors, via automatic sharding and replication. Moreover, ElasticSearch provides comprehensive data indexing, filtering, aggregation, and query support, which greatly simplifies the implementation of our high-level data processing algorithms. Our APM interfaces to the ElasticSearch service integrated within (if available) or external to the PaaS.

Logstash facilitates data extraction from a wide range of standard log formats (e.g. Apache HTTPD access logs). In addition, custom log formats are supported via simple configuration. Kibana provides a powerful web-based, dashboard with customizable data visualization. Kibana provides a wide range of charting, tabulation, and time series support. Finally, ElasticSearch (our data storage and processing engine) is easily integrated within popular "big data" workflows such as Spark and MapReduce [11, 27, 28] for advanced analysis.

Our Cloud APM implementation extends and customizes the default Elastic Stack deployment with agents and sensors across a PaaS stack. In addition, we add custom data processing and analytics components, as well as extensions that tailor the visualization of data to the metrics under interrogation (e.g. to make it easier to extract actionable insights).

4. APM USE CASES

We next describe a series of concrete use cases that we pursue with our PaaS APM. Our use cases employ the PaaS performance data collected by our APM to provide new PaaS features. In particular, we discuss how we use our APM system to predict performance-based service level objectives (SLOs) for the web applications deployed in a PaaS cloud, and how we detect performance anomalies across the PaaS stack.

4.1 Application Response Time Prediction

Our goal with this APM use case is to to provide scalable and accurate response time predictions that can be used between a cloud provider and PaaS user as a per-application SLO. To enable this, we combine static program analysis of the hosted web applications and APM monitoring of the PaaS cloud. Because we want to provide the prediction to PaaS users when they are deploying the applications, we perform this static analysis immediately prior to deploying or running an application on the PaaS cloud (as part of the PaaS application upload process) [13]. That is, we inject this new analysis component into the PaaS deployment process for cloud applications.

Our static analysis extracts, for each path through a function, the list of PaaS kernel calls (invocations and access to PaaS services) that are made. To enable this, we employ traditional techniques for abstract-interpretation-based loop bounds analysis, branch prediction, and worst case execution time analysis. We do not instrument the application to collect performance metrics at runtime (which would lead to the introduction of significant overhead). Instead we record these lists of calls in the APM system and monitor the services (i.e. the targets of these calls) in the system (independently from the application's execution).

In particular, our system employs the performance data from collected by the APM about PaaS kernel services. We implement an analysis routine for the APM that extracts a time series of operation execution times (response times) for each of the services of interest (e.g. that are in the list extracted from the static analysis of an application). We apply a forecasting methodology to calculate statistical bounds on the response time of applications. These forecasted values are then used by the cloud provider as the basis for a performance SLO [13, 14].

To make SLO predictions, we employ Queue Bounds Estimation from Time Series (QBETS) [23], a non-parametric time series analysis method that we developed in prior work. We originally designed QBETS for predicting the scheduling delays of batch queue systems used in high performance computing environments. We adapt it for use "as-a-service" in our PaaS APM system to predict the response time of deployed applications.

A QBETS analysis requires three inputs:

1. A time series generated by a continuous experiment,

2. The percentile for which an upper bound should be predicted ($p \in [1..99]$)

3. The upper confidence level of the prediction ($c \in (0, 1)$)

QBETS uses this information to predict an upper bound for the p-th percentile of the input time series. The predicted value has a probability of $0.01p$ of being greater than or equal to the next data point that will be added to the time series by the continuous experiment. The upper confidence level c serves as a conservative bound on the predictions. That is, predictions made with an upper confidence level of c will overestimate the true percentile with a probability of $1 - c$. This confidence guarantee is necessary because QBETS does not determine the percentiles of the time series precisely, but only estimates them.

As an example of how this process works, assume a continuous experiment that periodically measures the response time of a system. This results in a time series of response time data. Suppose at time t, we run QBETS on the time series data collected so far with $p = 95$ and $c = 0.01$. The prediction returned by QBETS has a 95% chance of being greater than or equal to the next response time value measured by our experiment after time t. Since $c = 0.01$, the predicted value has a 99% chance of overestimating the true 95th percentile of the time series.

In our APM processing plug-in, QBETS takes the response times for each PaaS kernel service we record in ElasticSearch. Note that this data is collected continuously by the PaaS monitoring agent, so QBETS is able to automatically adapt to the changing conditions of the cloud. Given the percentile for which an upper bound should be predicted and the upper confidence level of the prediction, QBETS can generate a conservative prediction.

Since an application may invoke multiple PaaS kernel services (those in the list for each operation generated by our static analysis at deployment time), the SLO predictor must also to align and aggregate multiple time series together before engaging QBETS. For example, suppose an application makes 3 PaaS kernel service invocations. The static analysis component would detect the 3 target kernel services invoked by the application. The SLO predictor then retrieves the response time data pertaining to those 3 PaaS kernel services from ElasticSearch. This information is retrieved as 3 separate time series. SLO predictor aligns the time series data by timestamp, and aggregates them to form a single time series where each data point is an approximation of the total time spent by the application upon invoking PaaS kernel services. Our analysis plug-in passes these aggregate time series in as the input to QBETS to make application response time predictions.

Note that our static analysis tool produces multiple sequences (per path) of PaaS service invocations for each analyzed application. The SLO predictor makes predictions for each of the paths identified by the static analysis tool. Our algorithm uses the maximum predicted value as the basis for a response time SLO between the PaaS and the application developer for the application under consideration.

Because PaaS service and platform behavior under load changes over time, our predicted SLOs may become invalid after a period of time. Our system detects SLO violations so that they may be renegotiated by the cloud provider. When such invalidations occur, the PaaS invokes our SLO analysis routine in the APM to establish new SLOs.

The key assumption that makes our approach viable is that PaaS-hosted web applications spend most of their execution time on invoking PaaS kernel services. Previous studies [13] have shown this to be true, with applications spending over 90% of their execution time on PaaS kernel service invocations.

We have integrated our Cloud APM within Google App Engine (as an App Engine app and without monitoring below the PaaS kernel) and have integrated it into the full stack of the private open source PaaS AppScale [16]. We use these platforms to perform extensive testing and empirical evaluation of open source Java web applications that execute over them [17, 14, 13]. We find that our system generates correct SLOs (predictions that meet or exceed their probabilistic guarantees) in all cases. Moreover, our results indicate that on average, the minimum duration for which our SLOs remain valid for this PaaS is 12 days. Our results also show that over a period of 112 days, the maximum number of times that any API consumer must renegotiate their SLA is 6.

4.2 Performance Anomaly Detection

Our second APM use case is the detection of performance anomalies. Numerous statistical models have been developed for detecting performance anomalies dynamically. However, most prior work focused on simple stand-alone applications. The goal of our work is to employ our PaaS APM to provide anomaly detection for PaaS-based (distributed) web applications.

To enable this, we implement multiple APM analysis plugins called anomaly detectors. Anomaly detectors are processes that periodically analyze the performance data for each deployed application in the PaaS. Our system supports multiple detector implementations, where each implementation uses a different statistical method to detect performance anomalies. We configure detectors at the application level making it possible for different applications to use 1 or more different anomaly detectors. Each anomaly detector has an execution schedule (e.g. run every 60 seconds), and a sliding window of data that it processes at a time (e.g. from 10 minutes ago until now).

In addition to multiple statistical anomaly detectors, we also implement a *path anomaly detector*. This detector leverages the PaaS kernel call list for each request processing path through an application – this is the same list that we employ for our SLO use case, which we can extract via static analysis, and that we describe in the previous section. However, in this work we use the data gathered from the PaaS kernel instrumentation (i.e. PaaS kernel invocation data) to infer the paths of execution for individual applications. This detector computes the frequency distribution of different paths and detects changes in this distribution over time. By doing so the path anomaly detector identifies the occurrence of new paths, most frequently executed paths, and significant changes in the path frequency distribution.

When an anomaly detector finds an anomaly in application performance, it sends an event to a collection of anomaly handlers. The event encapsulates a unique anomaly identifier, timestamp, application identifier and the source detector's sliding window that correspond to the anomaly. Anomaly handlers are configured globally (i.e. each handler receives events from all detectors), but they can be configured to ignore certain types of events. Like for detectors,

our APM supports multiple anomaly handlers – e.g., one for logging anomalies, one for sending alert emails, one for updating the dashboard, etc.

Additionally, we provide two special anomaly handler implementations: a workload change analyzer and a root cause analyzer. The workload change analyzer analyzes the historical workload trends of the applications to check if a particular anomaly is correlated with a recent change in the workload. This is done via a suite of change point detection algorithms on the workload data captured by the APM.

The root cause analyzer evaluates the historical trend of PaaS kernel calls made by the application, and attempts to determine the most likely components of the cloud (in the PaaS kernel) that may have attributed to a detected anomaly. To enable this we employ a combination of techniques that include relative importance for linear regression [10] and percentile analysis [23].

Both the anomaly detectors and anomaly handlers work with fix-sized sliding windows. They discard old data as the sliding window moves along the time line. As such the amount of state information these entities must keep in memory has a strict upper bound. Doing so makes these processes lightweight. Historical data can be persisted in the APM for offline batch processing if needed.

5. RELATED WORK

Application Performance Monitoring has become a fundamental requirement for any cloud platform. To date, there have been many monitoring frameworks designed and implemented to support gathering and analyzing performance data in order to draw insights about system behavior, performance, availability, and faults [21, 24, 26, 29, 22, 5, 7]. While many support data collection, storage, analysis and visualization to varying degrees, none of them are designed to operate as part of a cloud platform. Their data storage mechanisms (schema and query system), APIs and configuration model are targeted at monitoring servers or applications as individual entities. They do not provide any support for end-to-end tracing of request flows in a larger system. Further, they are not easily extensible, they support only basic metric calculations, and they provide no support for correlation or root cause analysis.

Anwar et al studied the monitoring facilities currently available in open source cloud platforms like OpenStack [1]. They showed that these frameworks use globally configured sampling rates for collecting data, and provide poor support for policy-based monitoring. We attempt to address these limitations by making it possible to configure agents and detectors at application level. In our system each can have its own sampling rate and monitoring policy. This allows fine tuning the monitoring support according to the needs of the individual user applications. Our system can also support changing sampling rates and monitoring policies for applications dynamically.

Dautov, Paraskasis and Stannett showed that a cloud platform monitor can be organized as a sensor network [6]. They argue that cloud platforms are dynamic (continuous change and evolution), distributed, have a high-volume of applications and data, and heterogeneous. To handle this complexity they propose instrumenting different components of the cloud with data collecting sensors. Sensors route data through a series of routing nodes into a central component that is responsible for storage and analysis. We follow a similar approach in which we instrument different layers of the PaaS cloud with sensors that report data to a central storage.

Corradi et al designed and implemented a integrated monitoring framework for the CloudFoundry PaaS [4]. This solution is organized into two modules – an availability monitor, and a performance monitor. The availability monitor uses periodic heartbeats to track the continuous operation of deployed applications. The performance monitor uses predefined application and database benchmarks to periodically evaluate application performance. While this solution is able to detect service outages and significant performance anomalies, it provides no support for workload change detection or root cause analysis.

Magalhaes et al have designed a series of systems for detecting performance anomalies in web applications [19]. Their work also addresses root cause analysis to some extent [18]. They use various statistical methods (correlation analysis, dynamic time warping etc) to detect anomalies in observed application performance. Then they look for any correlations between detected performance anomalies and workload level. If there are none, they attempt to perform root cause analysis. However, their solution requires instrumenting application code, so that backend API calls (e.g. database calls) can be intercepted and timed. To further enable this they also require implementing the applications using an aspect-oriented programming (AOP) framework. Provided that the application meets these requirements, their system is able to pinpoint the bottleneck backend API calls using a linear regression model.

We incorporate similar statistical methods into our APM processing components, but unlike their system, our approach does not instrument application code nor does it requires the application to be developed using a specific framework such as AOP. Their root cause analysis method also assumes that backend API call performance is independent, and shows no correlations (i.e. no multicollinearity). While this might be a reasonable assumption for classic web application deployments, in the cloud where the underlying platform is shared this assumption may not always hold. We therefore improve on their root cause analysis technique by using regression models that are resistant to multicollinearity [10].

Change point analysis plays a significant role in detecting changes in application performance and workload. This is a well understood area of statistics and we use a number of well known methods [15, 3, 2] in the implementation of our APM processing plug-ins. A key differentiator of our APM is our support multiple, concurrent statistical methods for both anomaly detection, workload analysis and root cause analysis thereby making it possible to compare, contrast, and combine different techniques.

Performance anomaly detection and bottleneck identification (PADBI) systems have been studied by a number of researchers in the past [12]. Unfortunately, the scale, multitenancy, complexity, dynamic behavior and the autonomy of cloud platforms make PADBI ill suited for PaaS cloud applications. By integrating monitoring and analysis within a PaaS, we believe that our approach overcomes these challenges. It does so by taking advantage of the scalable and fault tolerant services built into the platform and by having full visibility into all the layers, components, and interactions of the cloud platform.

6. CONCLUSIONS

As PaaS use grows in popularity, the need for technologies to monitor and analyze the performance and behavior of deployed applications has become fundamental PaaS features. However, most PaaS clouds available today do not provide adequate support for lightweight, full-stack performance data collection and analysis. Therefore, we propose a new application platform monitoring (APM) system that is able to take advantage of PaaS cloud features, while being easy to integrate within them.

To provide comprehensive, full stack monitoring and analytics, the APM we propose provides four major functions: data collecting, data storage, data processing, and data visualization. We describe the organization of our APM functions and identify the technologies that we use to implement them scalably. We show how our APM can be integrated into a PaaS system by customizing a set of API calls, and detail two use cases for APM processing that facilitate inference and prediction. Such functionality can be used to guide new PaaS services including response time SLOs at application deployment time, system wide performance anomaly and workload change point detection, and root cause analysis for application performance anomalies.

7. REFERENCES

[1] A. Anwar, A. Sailer, A. Kochut, and A. R. Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Asia-Pacific Workshop on Systems*, 2015.

[2] F. M. Bereznay and K. Permanente. Did something change? using statistical techniques to interpret service and resource metrics. In *Int. CMG Conference*, pages 229–242, 2006.

[3] L.-M. L. Chung Chen. Joint estimation of model parameters and outlier effects in time series. *Journal of the American Statistical Association*, 88(421), 1993.

[4] A. Corradi, L. Foschini, S. Fraternale, D. J. Arrojo, and M. Steinder. Monitoring applications and services to improve the cloud foundry paas. In *IEEE Symp. on Computers and Communications*, 2014.

[5] Datadog - Cloud-scale Performance Monitoring. `http://www.datadoghq.com` [Accessed April 2016].

[6] R. Dautov, I. Paraskakis, and M. Stannett. Towards a framework for monitoring cloud application platforms as sensor networks. *Cluster Computing*, 17(4), 2014.

[7] Application Performance Monitoring and Management - Dynatrace. `http://www.dynatrace.com` [Accessed April 2016].

[8] Elastic Stack. `https://www.elastic.co/products` [Accessed May 2016].

[9] App Engine - Run your applications on a fully managed PaaS. `https://cloud.google.com/appengine` [Accessed March 2015].

[10] U. Groemping. Relative importance for linear regression in r: The package relaimpo. *Journal of Statistical Software*, 17(1), 2006.

[11] Hadoop and Elastic Stack. `https://www.elastic.co/products/hadoop` [Accessed May 2016].

[12] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1), 2015.

[13] H. Jayathilaka, C. Krintz, and R. Wolski. Response Time Service Level Agreements for Cloud-hosted Web Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[14] H. Jayathilaka, C. Krintz, and R. Wolski. Service-level agreement durability for web service response time. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[15] R. Killick, P. Fearnhead, and I. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500), 2012.

[16] C. Krintz. The appscale cloud platform: Enabling portable, scalable web application deployment. *Internet Computing, IEEE*, 17(2):72–75, March 2013.

[17] C. Krintz, H. Jayathilaka, S. Dimopoulos, A. Pucher, R. Wolski, and T. Bultan. Cloud platform support for api governance. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, 2014.

[18] J. a. P. Magalhães and L. M. Silva. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[19] J. P. Magalhaes and L. M. Silva. Detection of performance anomalies in web-based applications. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, 2010.

[20] Microsoft windows azure. `http://www.microsoft.com/windowsazure/` [Accessed March 2015].

[21] Nagios - The Industry Standard in IT Infrastrucure Monitoring. `http://www.nagios.com` [Accessed April 2016].

[22] Application Performance Monitoring and Management - New Relic. `http://www.newrelic.com` [Accessed April 2016].

[23] D. Nurmi, J. Brevik, and R. Wolski. QBETS: Queue Bounds Estimation from Time Series. In *International Conference on Job Scheduling Strategies for Parallel Processing*, 2008.

[24] The OpenNMS Project. `http://www.opennms.org` [Accessed April 2016].

[25] SearchCloudComputing, 2015. `http://searchcloudcomputing.techtarget.com/feature/Experts-forecast-the-2015-cloud-computing-market` [Accessed March 2015].

[26] Shinken Monitoring. `http://www.shinken-monitoring.org` [Accessed April 2016].

[27] Spark and Elastic Stack. `https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html` [Accessed May 2016].

[28] Streaming and Elastic Stack. `http://thenewstack.io/building-streaming-data-hub-elasticsearch-kafka-cassandra/` [Accessed May 2016].

[29] Zabbix - The Enterprise-class Open Source Network Monitoring Solution. `http://www.zabbix.com` [Accessed April 2016].